Towards a Lazier Symbolic Pathfinder*

Benjamin Hillery
Brigham Young University
Provo, UT

Eric Mercer
Brigham Young University
Provo, UT

Neha Rungta
NASA Ames
Mountain View, CA

Suzette Person
NASA Langley
Hampton, VA

ABSTRACT

To explore the state space of programs with complex user-defined data structures, most symbolic execution engines use the lazy initialization algorithm. Symbolic Pathfinder (SPF) is the symbolic execution engine for the Java PathFinder (JPF) model checker; SPF too contains an implementation of the lazy initialization algorithm. A number of extensions to the original lazy initialization algorithm have since been published. One such extension is the lazier# algorithm which demonstrated dramatic performance gains over the other algorithms. There is, however, no open-source implementation of the lazier# algorithm available. This work is an implementation of the the lazier# algorithm within the Symbolic PathFinder framework. In addition, this work describes the implementation of two heap bounding techniques in SPF, namely k-bounding and n-bounding. The purpose of this paper is to discuss the nature of the improvements, implementation details, usage and performance test results.

1. INTRODUCTION

Symbolic execution is an automated program analysis technique that explores program execution paths using symbolic input values in lieu of concrete inputs [2, 9]. Symbolic execution constructs a path condition consisting of constraints over symbolic inputs that characterizes a program execution path. Off-the-shelf decision procedures and SMT solvers are used to check the satisfiablity of the path condition. The set of path conditions computed by symbolic execution is used to enable various verification, analysis, and testing tasks. Initially, symbolic execution largely focused on checking properties of programs with inputs of primitive types, such as integers and booleans; this was partly due to the extensive support provided by the solvers for checking satisfiability of constraints over these types. In order to symbolically execute object-oriented programs that are typical of Java, it is important to handle non-primitive types such as user-defined data structures and arrays. Several recent projects generalized the core ideas of symbolic execution to enable it to be applied to programs with more general user-defined types, including references and arrays [1, 3, 5, 7, 8, 12].

Symbolic Pathfinder (SPF) is the symbolic execution extension for the Java PathFinder model checker (JPF) [10, 11]. In order to handle symbolic execution of programs with complex data structures, it contains an implementation of the lazy initialization algorithm [8]. Whenever a symbolic object is first read along a path during symbolic execution, lazy initialization in SPF creates the following concrete choices for the object: (i) null, (ii) new instance of the object, and (iii) references to objects of the same type that were lazily initialized along the same execution path. Lazy initialization can often lead to a path explosion problem due to the concretization of the choices

Lazier algorithms that surpass lazy initialization delay object initialization deeper into the symbolic execution tree [3, 5]. This work is an implementation of the lazier# algorithm within SPF to more efficiently analyze programs with complex data types. This work also includes an implementation of k-bounding and n-bounding to better control termination in symbolic execution. The k-bound limits the length of the reference chains generated by lazy initialization, and the n-bound limits the total number of symbolic objects created by lazy initialization.

The paper is organized as follows: it first presents an overview of the lazier# algorithm and its implementation in SPF, and then gives a description of the implementation of the k-bounding and n-bounding techniques. Finally the paper compares the performance of the SPF implementation of the lazier# algorithm with that of lazy initialization using the same k and n bounds. That is followed by a discussion that compares our empirical results with those published in [3, 5]. The paper ends with a discussion on the related work, conclusion and an overview of future work.

2. LAZIER#

Deng et al. [3] introduced the lazier# algorithm and implemented it as a part of the Bogor/Kiasan framework to improve the performance of generalized symbolic execution.

*Research funded by NASA and NSF grant CCF-1302524

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

```
procedure LOAD(object r, field f)
    if r is initialized then
        if f has a reference type then
            assign f a new reference
        end if
        if f has a primitive type then
            assign f a new symbol
        end if
    end if
    if r is non-null then
        nondeterministically assign to r':
          an initialized version of r
          an existing location
        return read(r', f)
    end if
    if r is uninitialized then
        nondeterministically assign to r':
          a non-null version of r
          the value null
        return read(r', f)
    end if
    return r.f
end procedure
```

Figure 1: The lazier# initialization algorithm.

The generalized symbolic execution technique generates a concrete representation of connected memory structures using only the implicit information from the program itself. In the original lazy initialization algorithm, symbolic execution explores different heap shapes by concretizing the heap at the first memory access (read) to an un-initialized symbolic object. At this point, a non-deterministic choice point of concreate heap locations is created that includes: (a) null, (b) an access to a new instance of the object, and (c) aliases to other type-compatible symbolic objects that have been concretized along the same execution path [8]. The number of choices explored in lazy initialization greatly increases the non-determinism and often makes the exploration of the program state space intractable.

The Lazier# algorithm reduces non-determinism by pushing certain non-deterministic choices further into the execution tree. In the case of a memory access to an uninitialized reference location, by default, no choice point is created. Instead, the read returns a unique symbolic reference representing the contents of the location. The reference may assume any one of three states: uninitialized, non-null, or initialized. The reference is returned in an uninitialized state, and only in a subsequent memory access is the reference concretely initialized.

Fig. 1 presents the algorithm for the Lazier# initialization in a *load* operation for an object. There are three bytecodes in Java that perform a *load* operation: `getfield`, `getstatic`, and `aload`. The load operation in Fig. 1 takes as input an object reference r and field index f as parameters. If r is initialized, then the algorithm returns the field value $r.f$. If r is non-null, then r is either initialized or replaced with an alias, creating a non-deterministic choice point. If r is uninitialized, then r is either set to non-null, or replaced with the null reference also creating a non-deterministic choice point.

Besides loads and stores, other operations may change the state of r. For example, a conditional operator may attempt to determine whether a reference is null or not. In such cases, lazier# initialization makes a non-deterministic choice about the nullity of the reference and applies the appropriate constraint to the path condition before continuing execution.

To illustrate the differences in the results computed by the lazy and lazier# algorithms, consider the example shown in Fig. 2. The swap method, shown in Fig. 2(a), swaps the first two nodes in a list and returns a reference to the head of the list. The (partial) symbolic execution trees for the lazy and lazier# algorithms are shown in Fig. 2(b) and (c) respectively. Each node in the tree reflects the change to the symbolic state corresponding to symbolic execution of the code the line number shown to the right of the tree. Symbolic execution in JPF is performed at the bytecode level, so we show multiple steps corresponding to a single source line when necessesary.

The left branch of each symbolic execution tree reprsents the case where the field `next` is *null*. The right branches in each tree reflect the changes to the heap resulting from symbolic execution of the code inside the `if` statement. The lazier# algorithm computes a single heap representation corresponding to the five heap configurations computed by lazy initialization by using intermediate symbolic references that delay creation of non-deterministic choices.

2.1 Implementation of Lazier# Semantics

The `InstructionFactory` class in JPF implements the concrete semantics of Java bytecode; while the `SymbolicInstructionFactory` is used to implement the symbolic execution semantics of Java bytecode. The `SymbolicInstructionFactory` inherits from the `InstructionFactory` class in JPF. The symbolic execution semantics are implemented in bytecodes that are required for symbolic execution, e.g., conditional branch bytecodes, to override the concrete semantics. Similarly, we have implemented a `LazierInstructionFactory` that inherits from the `SymbolicInstructionFactory`; it overrides the bytecodes relevant to implementing the lazier# algorithm. The main bytecodes that are redefined in the `LazierInstructionFactory` are the load and store operations of complex data structures, e.g., `getfield`, `putfield` and conditional branch statements that check for null, e.g., `ifnull`.

In SPF, the symbolic values of variables are stored as attributes attached to the concrete values used by all JPF bytecodes. The attributes serve as signals to the relevant bytecodes to trigger symbolic execution specific behavior. During lazy initialization in SPF, the symbolic complex data structure references are assigned a symbolic value in their corresponding attribute at the start of the method. Once the object is concretized, the attribute for the corresponding object is set to `null`. In our lazier# implementation initialization, we add two different attribute types: *Symbolic Reference* and *Symbolic Location* symbols in order to distinguish between the phases of initialization as shown in Fig. 1. We also extended the `HeapChoiceGenerator` in order to support generating choices based on the lazier# algorithm.

The lazier# initialization algorithm requires renaming symbolic variables as they reach different stages of initialization. For example, a symbolic reference turns into a symbolic location. In order to handle initialization of symbolic locations without re-writing existing constraint equations, we created a listener to watch the stack for outdated symbols.

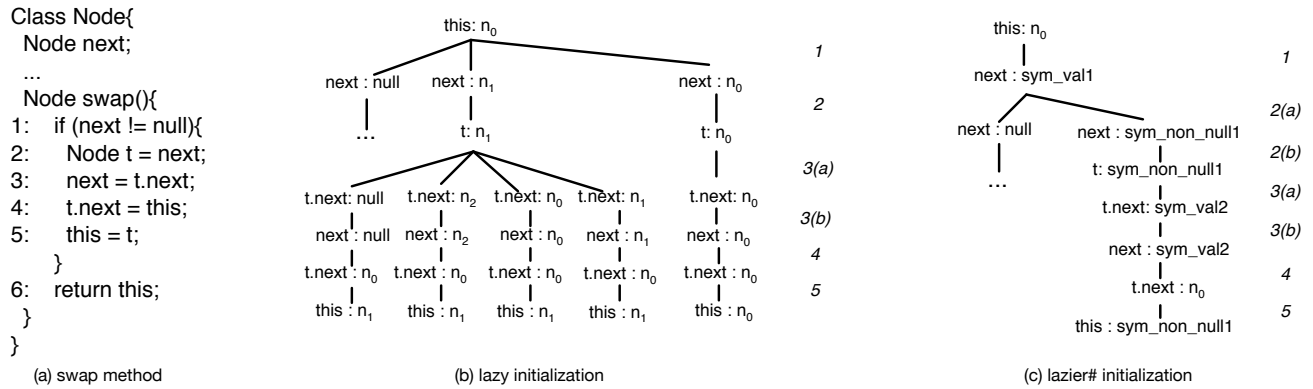

Figure 2: Comparison of lazy and lazier# algorithms.

The listener scans bytecode operand attributes for outdated symbols and renames them prior to bytecode execution.

3. BOUNDING

SPF currently supports bounding the search by a user-defined depth. This bound enables the search to terminate in the presence of unbounded loops. There is, however, a need for different bounding mechanisms in the presence of complex data structures. For example, we may want to guide the search towards longer reference chains. While certain heap properties may be controlled via preconditions written into the test program, it is cumbersome to make preconditions that apply only to the input heap as generated by lazy initialization, especially in the presence of destructive heap updates. Also, since the search depth includes choice generators for conditionals in addition to heap initializations, it is difficult to ensure even coverage of object references for a given depth bound.

In order to better facilitate checking programs with complex data structures there are two forms of bounding related specifically to controlling input heap structures: k-bounding and n-bounding. First used in Kiasan [3], k-bounding limits the length of reference chains produced by lazy initialization. At the beginning of the program, all static locations and reference parameters are initialized to depth zero. As program execution proceeds, accesses to objects at locations of depth m (from the root of the heap) generate lazy initializations to locations at depth $m + 1$. This continues until some maximum depth k is reached.

To illustrate the process of k-bounding for $k = 2$. Suppose we have an object A at depth 0, which contains a reference field. Note that depth 0 represents the root of the heap. To access the location referenced by that field in A, lazy initialization creates (i) a new object B at depth 1, (ii) an alias to A, and (iii) a null reference as illustrated in Fig. 3. If a subsequent operation needs to initialize a reference field of object B, we have similar choices as before: (i) create a new object C at depth 2, (ii) two aliases to objects A and B, and (iii) a null reference. Since object C is instantiated at depth 2 which matches the k-bound, the reference fields of C cannot be instantiated to the *new* choice. Fields of object C can only be initialized to aliases or the null reference.

The k-bound applies to the symbolic input heap, hence,

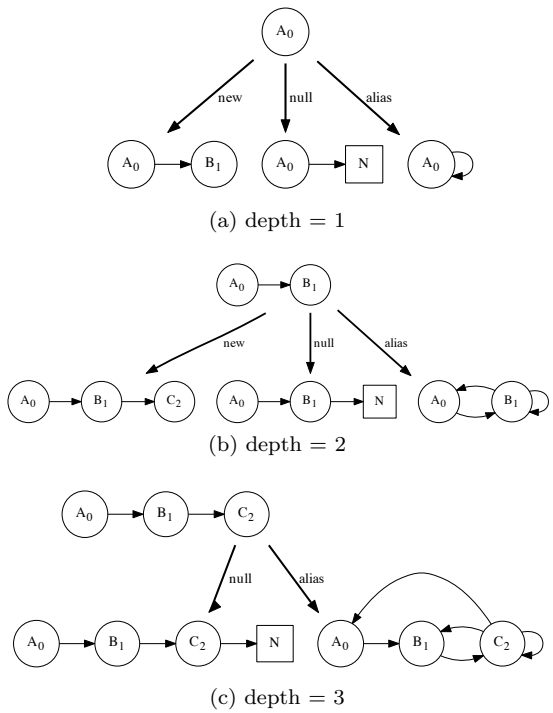

Figure 3: Example demonstrating k-bounding

the object depths remain fixed from the time of initialization, even if subsequent operations move them to shallower portions of the heap. For example, suppose a sequence of operations swaps objects B and C from Fig. 3. The depth associated with node C remains 2 despite the node's apparent position at depth 1. Any remaining reference fields in object C will be subjected to initialization rules for depth 2. k-bounding is useful for ensuring complete exploration of all heap shapes up to a given initialization depth. This even coverage provides a convenient way of comparing the efficiency of different lazy initialization algorithms.

In contrast to k-bounding, n-bounding trades complete-

ness of coverage for greater heap depth. n-bounding limits the total number of lazily initialized symbolic objects. n-bounding will explore much longer chains in a given amount time than k-bounding, at the expense of completeness.

The two bounding techniques are complimentary and can be used in tandem to adjust for intermediate depth/breadth tradeoff levels. Note that k-bounds and n-bounds apply only to the input heaps presumed to exist at the start of the method, and do not limit the size of static or dynamic memory structures created by the method itself. In other words, these bounds do not apply to static objects created prior to method start-up, objects created by the `new` operator, or objects eliminated by garbage collection. Thus, the total number of active heap nodes may appear to exceed or fail to reach specified bounds. If limitations on absolute heap size are desired, they may be enforced by means of method pre/post conditions.

3.1 Implementation of Bounding

To implement k-bounding in SPF, additional depth information is stored in an attribute object attached to the corresponding `ElementInfo` class of the object. Note that information about objects are stored in JPF within the `ElementInfo` class. For bounding, a listener monitors the total number, or depth, of initialized objects in any instance of a `HeapChoiceGenerator`. Recall that the `HeapChoiceGenerator` creates the points of non-determinism for the lazy initialization algorithm. We have implemented the corresponding choice generators for the lazier# algorithm as well.

Both k-bounding and n-bounding ignore heap structures larger than the specified bounds by treating them as if they are infeasible. If a bytecode attempts to access an object along an execution path that exceeds the search bounds, the state is marked as ignored and the JPF virtual machine backtracks to the previous choice generator.

4. RESULTS

In this section we present results of our Lazier# algorithm, k-bounding and n-bounding techniques. All our experiments are conducted on a 1.83GHz Intel Core 2 Duo processor and 2GB RAM. JPF v7 was used as the software platform, and the Lazier# implementation was based on the `jpf-absinth` implementation of the Symbolic PathFinder[1].

Method	k	Time		States	
		LZ	LS	LZ	LS
TreeMap put	1	0:02	0:02	148	46
	2	0:13	0:02	6622	446
	3	0:19	0:09	25795	10504
TreeMap get	1	0:01	0:01	40	64
	2	0:04	0:03	1654	958
	3	1:07	0:25	79045	27569
BST insert	1	0:01	0:01	27	25
	5	0:05	0:04	303	145
	7	4:39	1:09	7113	1705
BST findMin	1	0:01	0:01	15	19
	2	0:03	0:03	143	143
	3	1:01	01:04	2373	2123

Table 1: k-bounding results

[1]jpf-absinth is a fork of jpf-symbc

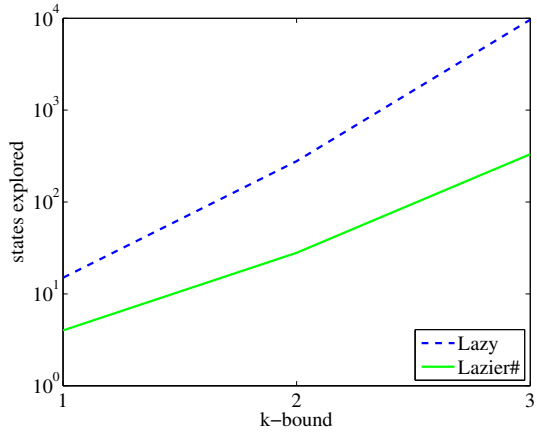

Figure 4: Treemap.put execution time

Method	n	Time	States	Paths
TreeMap put	2	0:03	258	14
	3	0:09	1750	28
	4	0:40	17058	64
	5	3:54	222535	152
	6	23:34	3630321	360
BST findMin	2	0:01	19	2
	3	0:02	80	4
	4	0:05	320	9
	5	0:24	1234	23
	6	2:11	4698	65

Table 2: n-bounding results

Methods from `java.util.TreeMap` and `BinarySearchTree` [13] are used in the k-bounding experiments. For these experiments, the n-bound and search depth parameters are set to a value of one million in order to ensure that the k-bound is indeed the limiting factor. All methods were executed using preconditions specified by [4], over k-bounds of 1,2, and 3. Table 1 shows the results of the experiments for the dependent variables of execution time, states explored, and number of valid terminal paths.

Figure 4 shows run-time for our implementation vs. the Kiasan implementations of Lazy and Lazier# initialization as reported in their paper. Since we cannot make claims about efficiency given that both the implementations are run on different machines, it is noteworthy that the trend of our implementation parallels that of Lazier# rather than that of the lazy initialization algorithm.

For the bounding experiments, each bound is isolated by setting the other bound to a large value that is unreachable in the experiment sets. Table 2 contains the results of n-bounds analysis of the `put` method in the `TreeMap` class and the `findMin` method in the `BinarySearchTree` (BST) class.

In all k-bounding experiments, the number of paths explored matches exactly with those that were reported in [4]. Since the correct k-bounding path results were established analytically [5], we have a reasonable degree of confidence in the correctness of the lazier# implementation when used in conjunction with k-bounding.

In the n-bounding experiments, path counts for the `put` method in the `TreeMap` class match the published results in

[4]. However, path counts for `BinarySearchTree.findMin` appear to differ from the published figures by one, i.e., our path count for $n = 6$ matches the count in [4] for $n = 5$. We are still investigating the cause for this discrepancy, which is curious considering results from other n-bounding experiments seem to match perfectly.

We also conducted some analysis of the lazy initialization implemented within SPF. However, the results did not correlate well with the published figures, and a number of tests failed to complete due to some bug in the current implementation in jpf-absinth. Since we have not established the cause of the difficulties, those figures are not included in this work. We are working on fixing the errors within the lazy initialization implementation.

5. RELATED WORK

Generalized symbolic execution [8] extends traditional symbolic execution to reason about open systems that manipulate dynamically allocated data, i.e., user-defined types. It implements a *lazy* initializatin algorithm that delays materialization of heap objects until a field in the object is first read. In [3], Deng et el. introduce k-bounded symbolic execution and present a *lazier* initialization algorithm, implemented as part of the *Bogor/Kiasan* framework. This *lazier* algorithm delays initialization of reference fields longer than the lazy algorithm in generalized symbolic execution by dividing lazy initialization into two steps. The first step occurs when an uninitialized reference type variable is read, at which point the variable is initialized to `null` or a fresh symbolic *reference value*. In the second step, this branch is split at the point when a field of the symbolic reference is accessed, i.e., read or written. At this point, the symbolic reference is non-deterministically replaced by choosing an existing type-compatible object or a fresh symbolic object. This approach delays the non-deterministic choice of objects in the lazy initialization algorithm, and in some case, the delay may be indefinite. In [5], Deng et al. introduced an even lazier algorithm, *lazier#* – the algorithm implemented in this work. *Lazier#* addresses the fact that the lazier algorithm optimistically assumes most variables are `nonnull`, by introducing an intermediate step that creates a new type of symbolic value which abstracts `null` as well as any object of the appropriate type. This has the effect of delaying the first branch point until a field in the symbolic value is accessed.

Recent work introduces a front-end analysis before symbolic execution to determine constraints on heap shapes up to a given bound to prune the number of instantiations considered in lazy initialization [6]. As a result, symbolic execution does not explore redundant isomorphic heap shapes. The front-end analysis has a high cost but yields significant reductions and is complementary to the lazier# algorithm.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an implementation of the lazier# algorithm in SPF. The lazier# algorithm uses fully symbolic objects to delay instantiating heap objects deep into the program execution. The result is that the algorithm generates many fewer states and explores many fewer paths as shown in a set of benchmark examples. The paper further describes the implementation of two distinct methods for bounding symbolic execution in SPF. The first method, n-bounding, caps the number of new instances in the symbolic heap. The second method, k-bounding, caps the depth of heap objects initialized in any single reference chain. JPF's annotation framework is used to implement both bounding techniques. Future work is to extend this notion of fully symbolic objects to a complete symbolic heap that does not initialize concrete heap objects. That is to say, objects are no longer instantiated but rather are represented as constraints similar to path constraints in symbolic execution.

7. REFERENCES

[1] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, pages 2–23, 2005.

[2] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[3] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[4] X. Deng, J. Lee, and Robby. Efficient and formal generalized symbolic execution. *Autom. Softw. Eng.*, 19(3):233–301, 2012.

[5] X. Deng, Robby, and J. Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. In *SEFM '07: Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 273–282, Washington, DC, USA, 2007. IEEE Computer Society.

[6] J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser. Bounded lazy initialization. In *NASA Formal Methods*, pages 229–243. Springer, 2013.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[8] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[9] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[10] C. S. Păsăreanu and N. Rungta. Symbolic Pathfinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[11] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–25, 2008.

[12] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272, 2005.

[13] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999.